# AN EDITOR TO SUPPORT MILITARY MESSAGE PROCESSING PERSONNEL

Jeff Rothenberg

University of Southern California

Prepared for:

Advanced Research Projects Agency

June 1975

ARPA ORDER NO. 2223

ISI/RR-74-27
June 1975

Jeff Rothenberg

# An Editor to Support Military Message Processing Personnel



INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>ISI/RR-74-27 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER<br>AD-A012 544 |
| 4. TITLE *(and Subtitle)*<br>An Editor to Support Military Message Processing | | 5. TYPE OF REPORT & PERIOD COVERED<br>Research |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Jeff Rothenberg | | 8. CONTRACT OR GRANT NUMBER(s)<br>DAHC 15 72 C 0308 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>USC/Information Sciences Institute<br>4676 Admiralty Way<br>Marina del Rey, CA 90291 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>ARPA Order # 2223<br>Program code #3D30 &3P10 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Advanced Research Projects Agency<br>1400 Wilson Blvd.<br>Arlington, VA 22209 | | 12. REPORT DATE<br>June 1975<br>13. NUMBER OF PAGES |
| 14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* | | 15. SECURITY CLASS. *(of this report)*<br>Unclassified<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT *(of this Report)*

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

---

18. SUPPLEMENTARY NOTES

---

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Automatic formatting, case-classes, change-categories, change-types, correcting, editing functions, intraline editing, placemark, replacing, restructuring, searching.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(OVER)

PRICES SUBJECT TO CHANGE

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Military message processing in an on-line environment requires the flexible tools for entering, editing, formatting, and annotating text provided by the Editor designed for the Information Automation project. The Editor allows users to read documents or messages on-line (providing suitable browsing and scanning facilities); to enter text (allowing simple correction of typographical errors); and to edit already existing text (in any of several ways, including style changes, spelling correction, reformatting and reorganizing, as well as adding annotations or suggestions). Inexperienced or occasional users can perform most editing tasks using a minimal set of simple commands, relying on terminal controls and function keys for the most frequent operations. More sophisticated users can perform powerful editing functions with a handful of additional commands. The Editor is closely geared to the tasks of message authorship and coordination; in particular, it addresses the issue of communication among multiple authors and coordinators by means of annotation facilities.

ARPA ORDER NO. 2223

ISI/RR-74-27
June 1975

Jeff Rothenberg

# An Editor to Support Military Message Processing Personnel

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO DAHC15 72 C 0308 ARPA ORDER NO 2223/1, PROGRAM CODE NO 3D30 AND 3P10

## *CONTENTS*

Preface *v*
Summary *vii*

1. Introduction *1*
    - Input Interface *1*
    - Low-Level Editing *1*
    - Full Editor *2*

2. Editor-Related Tasks *3*
    - Read *3*
    - Enter Text *4*
    - Modify/Edit Text *5*
        - Style (or Rewrite) Changes *6*
        - Spelling Corrections *7*
        - Format Changes *8*
        - Restructuring *8*

3. Basic Editing *9*
    - Read *9*
        - Move Cursor *9*
        - Find (String) *10*
        - Placemark *11*
    - Input Interface (Corrector) Compatibility/Enter *12*
        - Editing *12*
            - Cursor Motion *12*
            - Text Modification *12*
            - Text Replacement *13*
            - Text Insertion *13*
        - Cursor Motion Commands *13*
            - Step Unit *13*
            - Direction of Motion *14*
            - Erase or Do Not Erase *14*
    - Cursor Editing (Correcting) and Style Changes *14*

4. Editing Commands 17
Replace 17
Spelling Corrections 19
Formatting 20
Restructuring 22
Special-Purpose Sub-Editors 23
Names and Titles 24
Dates 25
Restricted Vocabulary 26

5. Issues 27
Coordination 27
Coordinator Options 27
Spelling Error 28
Style (Rewrite) 28
Structure (Format) 28
Minor 28
Content 29
Crucial 29
Examination Options (Viewing Changes) 29
Spelling Error 30
Style (Rewrite) 30
Structure (Format) 30
Formatting 31
Correcting (Cursor Editing) 31
Viewing Coordination/Edit Changes 32
Summary 32

Appendix
A. Editor Commands 33
B. Terminal Considerations 49

References 53

Bibliography 55

# *PREFACE*

This report is one of a planned collection of reports describing the current status and plans of the Information Automation project. It is intended to be read by members of the Advanced Research Projects Agency (ARPA), Computer Science specialists, and medium- and high-level military personnel.

The Information Automation (IA) project [1] is currently developing methods to automate various information handling tasks, with particular emphasis on message processing for military command, control, and communications [2]. The project is sponsored by ARPA, and is an integral part of both the client's and ISI's overall program to explore the use of computer technology and methodology in military environments.

Other project elements are referred to where appropriate, but are not defined herein, since they are described in detail elsewhere. Because the Input Interface (or Corrector) of the Command Language Processor (CLP) is logically the lowest level of the Editor, the CLP document [3] should be read in conjunction with this report. In addition, the Editor facilities are strongly motivated by the message processing service, so Ref. 4 should also be read as background for the present report.

For a more comprehensive discussion of other project elements, the reader is referred to project documentation noted in the references.

## *SUMMARY*

Military message processing in an on-line environment requires flexible tools for entering, editing, formatting, and annotating text.

The IA Editor supports these functions pervasively throughout the service. It is always available when the user is dealing with text, whether in the form of commands or messages. It allows users to

- Read documents or messages on-line (providing suitable browsing and scanning facilities).
- Enter text (allowing simple correction of typographical errors).
- Edit already existing text (in any of several ways, including style changes, spelling correction, reformatting and reorganizing, as well as adding annotations or suggestions).

The Editor allows inexperienced or occasional users to perform most editing tasks using a minimal set of simple commands, relying on terminal controls and function keys for the most frequent operations. More sophisticated users can perform powerful editing functions with a handful of additional commands.

The Editor is closely geared to the tasks of message authorship and coordination. In particular, it addresses the issue of communication among multiple authors and coordinators by means of annotation facilities.

# 1. INTRODUCTION

The Information Automation (IA) project is concerned with the creation, management, dissemination, storage, and access of text. In particular, it provides facilities for manipulating, editing, and annotating text, formatting messages and documents, reading and scanning soft copy, and correcting input, whether commands or text, as it is being typed. It also provides multi-author text creation and coordination, commenting, and annotation. Since most of these functions must be available whenever a user is dealing with text, they are concentrated in a single module called the Editor, which is always available as part of the interface rather than being dispersed among different program elements.

Editing may be viewed at three logical levels: the Input Interface, the low-level Editor, and the full Editor.

## *INPUT INTERFACE*

This provides intraline editing of the user's input, whether commands or text. This is provided by the Command Language Processor (CLP) and is carefully human-engineered to be as natural as possible (the actual interface varies depending on the terminals used). The Input Interface (which is logically the first level of the Editor) requires absolutely minimal training and has the flavor of actually "making corrections" to the text rather than "issuing commands" to the Editor.

## *LOW-LEVEL EDITING*

Because the intraline capabilities of the Input Interface are generalized to extend to text in general (by means of the ability to move up and down through multiline text as well as horizontally within a single line), the user is given the facility to perform most editing tasks ***without*** the need for actual commands. The Input Interface is already engineered to be natural, so this low-level Editor requires essentially no additional training. It does not provide the full power of searching, moving blocks of text, or replacing strings of text with other strings, but it does allow a beginning user to edit text as soon as he starts using the service.

This level of the Editor also includes facilities to read text (browse and scan) and to attach comments and annotations to documents.

## *FULL EDITOR*

This includes searching, replacing, formatting, block manipulation (moving paragraphs of text or rearranging sections of a document), multi-author, and full annotation capabilities. It minimizes the number of distinct commands without going to the opposite extreme of complex parameterization. Appendix A summarizes the actual commands.

## 2. EDITOR-RELATED TASKS

This section discusses those tasks the user performs with the Editor. The functions introduced here are discussed in later sections in greater detail.

The user has three primary classes of tasks for the Editor:

- Read
- Enter text
- Modify/edit text

The Editor functions are introduced in terms of these tasks.

### READ

The Editor provides a protected Read capability which allows reading and moving around in text without modifying it. This is used for reading delivered messages or for scanning documents without modifying them.

The implicit command in this mode is "Show," which displays text in its fully formatted output form. Normally, a full page of text is shown, though it may be necessary to allow looking at smaller contexts while moving around in the text.

The user has basic "moving" functions which move a pointer that locates "where the user is" in the text (hereafter, this pointer is not distinguished from the cursor that appears on the user's terminal, which shows him his place in the text on the screen). The default is to show some text above and below the cursor, and this total context is called the "display window." Only vertical cursor movements are enabled in this mode, the basic ones being

- Move cursor to the top of the text
- Move cursor to the bottom of the text
- Move cursor up one screenful
- Move cursor down one screenful

with additional optional functions:

- Move cursor up one "chunk" of text
- Move cursor down one "chunk" of text

("Chunk" is defined in terms of the structures used by the target community--for example, it might be "paragraph", or "section". This allows fast browsing through text.)

The "up" and "down" functions are geared to the size of the screen to provide a single-page movement with enough overlap to maintain context, e.g., two lines on a twenty-line screen. Alternatively, half-page moving may be better for reading on certain screen sizes.

Two additional moving functions are allowed for somewhat more sophisticated users:

- FIND text
- PLACEMARK

Finding text involves a restricted version of the Search facility provided under Modify/Edit (see below), consisting of the simple form "FIND <string>". This produces an exhaustive search through the text for <string>. (Note that here the service-wide commands UNDO and REDO can be used to make the user's job easier. The FIND command displays the text found (if any) and moves the cursor to it; to reject this move and return to the last position in the text, the user can simply UNDO. Similarly, to find the next occurrence of <string> without having to retype the FIND command, the user can REDO.)

The placemark is an optional feature useful primarily when dealing with large bodies of text. It allows the user to insert placemarks and later return to the marked place in the text. A PLACEMARK command is provided, which produces a temporary mark (lasting only as long as the user is working on this text), and a FINDPLACE command allows returning to a previous placemark. These are described in detail in Section 3.

### *ENTER TEXT*

Whether he is creating a new document, commenting on an existing one (as a reader), or inserting additional text when editing a document, the user always enters text in the same way. Even at the simplest level of entering commands to the service, the Input Interface (or Corrector) provided by the CLP offers the basic facilities for correcting text as it is typed in. (The Input Interface is described in detail in Ref. 3, but its capabilities are described in the following section for completeness.)

When entering text, the user is able to back up easily into the text to correct typographical errors. He is not required (though he is allowed) to first terminate entering text, then use the editing commands to correct simple errors or rewrite what he has typed. He can back up the cursor without erasing what he has already typed, insert, delete, and overstrike, and return to the "end" (the last thing he had typed at the end of the text). (Note that this "end" may not be defined except in the simple case where the user is creating a new document or text field which is clearly not yet finished. When the Editor cannot resolve where the "end" is, this command simply causes a bell, and the user must reposition the cursor himself.)

When moving around in the text being inserted, the user can move backwards or forwards within lines, with or without erasing the text that the cursor passes over. He can move the cursor a character, word, or line at a time. Since he can erase with the cursor, he can thus erase a character, word, or line at a time.

The Input Interface editing functions are a subset of those provided by the Editor. The Editor is upward-compatible from the Input Interface. This service-wide consistency provides uniformity and ease of use throughout.

Note that whether the user is entering text or editing, the Editor protects him against losing more than a few minutes of work in the event of system failure. This is done by checkpointing the text (off to a secondary storage medium, e.g., disk). Several conditions trigger checkpointing to insure that large amounts of text (anything greater than one screenful) cannot be lost, and that complex editing sequences need not be redone should the system crash.

### *MODIFY/EDIT TEXT*

In order to make extensive or after-the-fact changes to text that has already been entered, the Editor provides the user with several classes of commands, corresponding to the kinds of changes he may want to make to text. These are

- Style (or rewrite) changes
- Spelling corrections
- Format changes
- Restructuring

These classes are described below. Later sections will return to the commands in each class and discuss them in greater detail.

*Style (or Rewrite) Changes*

This class includes many of the hard-to-categorize minor editing changes that are made on paper with pencil and eraser. An example would be editing the phrase

"functions which move the pointer into the buffer"

to read

"functions which move a pointer that locates 'where the user is' in the text".

This class is comprised of operations for inserting and deleting (similar to those provided in the Corrector) and for replacing.

There is almost no difference between correcting text that is being typed in and changing text already entered. In the case of entering input, the user is already in the "Edit mode" which allows him to position the cursor and make changes by means of the Corrector (Input Interface). In the case of editing (modifying) existing text, the user must first enter the "Edit mode" and then make his modifications. This is done with an "Edit switch" (or command). The Edit mode is also indicated on the display (see Section 5).

The cursor controls in this mode allow "fine" movement within pages and lines. In particular, as for the Input Interface, the cursor can be backed up without deleting so as to change (or delete, or insert) something earlier in the text, without having to retype the intervening text.

The Corrector functions allow the editing of a line as it is being typed. The fine vertical cursor movements (combined with the gross ones already described above under Read functions) allow the same techniques to be applied to editing existing multiline text. The beginning user normally edits text by beginning in Read mode, moving around till he finds something to be changed, switching into Edit mode, zeroing in on the spot to be changed by moving the cursor, correcting it just as he would have when first entering it, and going back into Read mode.

Though the single insert function suffices for all but the most sophisticated types of insertion (such as moving blocks of text around--which is provided by additional commands described below), "Delete Character" is not sufficient even at the lowest levels, and is supplemented by two (or three) other Delete functions: Delete Word and Delete Line are identical to the Corrector deleting capabilities described in Section 3.

There is also an optional "Delete <Structure>," where <Structure> is defined for the target community as required--paragraph, page, etc.

All Deletes are shown on the display, and can be undone by UNDO.

Style changes may also require the use of the REPLACE command. In its simplest form, this is a powerful and easy-to-learn command which allows the user to "REPLACE <string1> BY <string2>". Like the FIND command mentioned above, the REPLACE can be UNDONE or REDONE. In particular, the user can easily iterate through a number of similar replacements, examining each one in turn, by means of REDO.

The combination of the Corrector (cursor moving, insert, delete, and overstrike) and the REPLACE command handles most of the editing tasks the user performs.

Though they are not used as frequently, the remaining classes of functions are indispensable, and greatly increase the user's control over the text he produces.

### *Spelling Corrections*

Though the REPLACE command already gives the user the capability to correct typographical or spelling errors, a special SPELLING command is provided which allows the user either to make single corrections or to cause all occurrences of a word in the text to have their spellings changed.

In addition to performing the replacement, this command marks these changes as Spelling Corrections (for later reference by coordinators or authors--see below, Section 4), and it alerts the User Monitor [5] which considers entering this spelling error into the service-wide lexicon of misspellings and typographical errors (see also Ref. 3).

### *Format Changes*

The basic approach to formatting is to attempt to provide reasonably formatted text without requiring that the user do ***anything*** explicit to produce formatting. This default Automatic Format mode is described in detail in Section 3. The only actual format commands available to the user are those which override or turn off the actions of the Automatic Formatter. These are

- Begin Literal (turns off Automatic Formatting in the text)
- End Literal (turns Automatic Formatting back on)
- Delete/Insert Paragraph Mark
- Delete/Insert Page Mark

The Literal feature allows the user to type text on his terminal in the format he wants to see on output, and to have it appear exactly as it did when he entered it (within the limitations of differing output devices). The Begin and End Literal allows this Literal Format mode to be turned on and off again several times within a given document. The parts of the document that are not under Literal Format mode are formatted automatically, as usual.

Though paragraphs and page boundaries are handled automatically by the Automatic Formatter, the user needs some way to modify this. The minimal set of commands: Delete/Insert <Paragraph Mark/Page Mark> provide the control necessary to create paragraphs and page boundaries explicitly when the user needs to do so. Most of the time, however, these commands are not needed.

### *Restructuring*

Occasionally the user needs to rearrange text (or instance by moving a paragraph to another place in the document, exchanging two sections, copying a sentence from one place to another, etc. These tasks are difficult to perform without a special command, since they would otherwise require large amounts of retyping.

The Editor provides the MOVE command to perform this task. The body of text to be moved can be selected in one of two ways, both of which use the cursor controls to point out a place in the text: either the user names the amount of text to be moved (Word, Sentence, Paragraph, etc.) and points the cursor anywhere in the text to be moved, or he uses the cursor to select the start and end of the text to be moved.

# 3. BASIC EDITING

This section describes the commands the user needs for most tasks. These provide a subset of the total capabilities of the Editor, and are sufficient for most editing jobs. They are designed to be few and simple to learn. They are supplemented by the commands described in Section 5, which deliver the full power of the Editor.

## READ

The reading facilities of the Editor have been sketched above (Section 2). Whenever the user is reading a document, whether his own, a delivered message, or a message which he is helping to write or coordinate, he is able to browse and scan through the text with three major functions:

- Move cursor
- FIND text
- PLACEMARK

These functions are all available to the user whether he is reading or modifying text. The only distinction is that when he is just reading (that is, when he is not in the Edit mode), there are certain restrictions placed on what he can do. This protects him from inadvertently changing text he only wanted to read.

The allowed functions for reading text operate as described in the following subsections.

### Move Cursor

This function is sufficient for reading text in most situations. A beginning user need not be concerned with FIND or PLACEMARK.

There are three ways the user can move the cursor through his text. Each can be applied either forward or backward.

- To the top or bottom of the text
- Up or down one screenful
- Up or down one "chunk"

The normal way to read text is to start at the top, where the Editor automatically places the cursor when the user requests to read a document. The user then proceeds downward one screenful at a time. The "up" and "down" movements rewrite a screenful, keeping a small amount of overlap to maintain context. That is, on a twenty-line screen, the "down" command will normally rewrite the last two lines of the previous screenful at the top of the screen, and add eighteen new lines at the bottom. This overlap is easily disabled or changed for a particular target community or user, being controlled by one of several Editor parameters stored for each user. (The service maintains a User Profile for each user, which contains information about his preferences for how the interface should work. For more detail on this, see Refs. 4, 5, and 6.) Similarly, overlap is maintained whenever the user moves in either direction to adjacent text. When one moves upward through the text, the overlap is displayed at the bottom of the screen.

The "chunk" Move commands are provided for browsing through large bodies of text, where one screenful at a time is too slow to locate the desired part of the text. The definition of "chunk" is entirely up to the user community. In general, it will depend on the kind of structure commonly present in the text to be dealt with, and also on the terminal and output device characteristics. For example, if reports with section numbers are frequently encountered, the "chunk" might look for the next section, either forward or backward in the text. On the other hand, if documents are intended for a hard-copy output device which produces pages (especially if these consist of several screenfuls of text as it appears on the terminal), then "chunk" can move up or down to the next page.

***FIND <string>***

This simple command allows the user to search for any <string> in the text. For instance, "FIND < ship >" looks for the word "ship", "FIND <.>" looks for a period, and "FIND < will be >" looks for the two words together. (Note that this is not meant to represent the actual command form for FIND, since the command language (see Ref. 4 and 5) is dependent on the target user community.)

The normal case is to search forward through the text, below where the user is pointing (that is, below the cursor). If <string> is not found by the end of the text, FIND then looks backwards toward the top of the text. This insures that FIND will always find <string> if it occurs anywhere in the text, and that it will find the closest occurrence to the cursor, first looking downward, then upward. The user can override this strategy in several ways: he can go to the top or bottom of the text before using FIND (to find the first or last occurrence of <string> in the text), he can explicitly ask FIND to look backwards first, or he can ask it to look ***only*** forwards ***or*** backwards.

The default for the FIND command is that the user is looking through text which he has seen formatted by the Automatic Formatter (see Section 4). Therefore, the user is not expected to worry about whether two words are separated by a space, a carriage return, a tab, etc. Normally, the user delimits the <string> to be found, as, say, < ward >, where the spaces around "ward" distinguish the fact the he is looking for "ward" and not "towards". (If a particular user finds this delimiting confusing, he can use the command FINDWORD, which automatically delimits <string>. The FIND command looks for "ward" in the middle of a line, at the beginning of a line, and at the end of a sentence (followed by a period). In addition, it looks for it at the beginning of a sentence, which requires looking for "Ward" as well. The strategy with respect to capitalization is that if <string> is typed by the user in lower case, it will be matched as "ward", "Ward", or "WARD". However, if the user typed it as "Ward" or "WARD", it is only matched as typed. Similarly, in the above example ("FIND < will be >"), occurrences with "will" and "be" separated by a carriage return are found, but "will. Be" is ***not*** considered a match.

All this matching strategy is designed to alleviate the need for the user to think about special cases. However, sometimes it gets in his way, and so the special FINDEXACT command is supplied, which matches only what the user types for <string>, exactly as he types it.

### *PLACEMARK*

When the user is actually editing text, he can create his own placemarks easily enough by inserting special characters in the text, and then searching for them to return to the same place. For example, the user might insert !rewrite this! in the body of a message, and might then return to it later with "FIND <!rewrite this!>". However, he runs the risk of leaving marks like this around unintentionally to clutter up the text. Further, this facility is not available unless he is in the Edit mode. He needs some similar facility when in the Read mode, which does not make permanent changes to the text. This is provided by the PLACEMARK and FINDPLACE commands.

PLACEMARK allows the user to make a temporary mark in the text wherever the cursor is positioned. (In the Read mode, this can only be at the start of a line. However, in the Edit mode, the cursor can be anywhere in the text, and the mark can be placed in the middle of a line.) These marks are always temporary, and disappear when the user finishes his editing session with this document.

In its simplest form PLACEMARK and FINDPLACE take no arguments. Every time the user does a PLACEMARK, the previous placemark is erased and a new one is created. Most of the time, this serves the user's needs fully, since he frequently just wants to

keep track of where he was in the text while he goes off to another place in the text to check something. FINDPLACE returns him to where he last was, and he keeps this place marked until he uses another PLACEMARK command.

Occasionally, the user may want to mark several places in the text and move among them, keeping them all marked separately at once. To allow this, the PLACEMARK and FINDPLACE commands actually allow an argument which consists of a name for the place marked. When the argument is omitted, as above, the null name (that is, no name) is used, and there is only one placemark available. However, when the user needs multiple placemarks, he can make up his own names for them. For example, he can use numbers, and say "PLACEMARK (1), FINDPLACE (3)", which marks his initial location in the text as PLACE (1), (erasing any previous placemark named "1"), and finds the place marked as PLACE (3). Alternatively, he can name places "A", "b", "John", "Mary", "important", etc.

All placemarks are temporary, lasting only for the duration of the editing session.

## *INPUT INTERFACE (CORRECTOR) COMPATIBILITY/ENTER*

Much of the user's low-level editing is done while inserting text. This makes use of the Corrector functions which allow backing up (with or without erasing) and making insertions, deletions, and minor changes without using actual Editor commands. These capabilities have already been discussed above (see Section 2).

The Input Interface, or Corrector (provided by the CLP), provides these editing functions when inserting any text, whether commands to the service, or text for the Editor. It is described here for completeness. For further discussion, see Ref. 3.

### *Editing*

The Input Interface's editing commands are all activated by single keystrokes. There are two classes of editing commands: the main class involves cursor motion and variations on cursor motion, and the second provides keystrokes to delimit new text.

***Cursor Motion.*** Each time the user hits one of the cursor motion keys, the cursor moves one unit. With the simple motion commands, the user has easy access to any part of his input text. The actual commands are discussed below.

***Text Modification.*** Once the user has moved the cursor to the position in the text at which he wishes to make a change, he needs facilities to actually change the text.

***Text Replacement.*** If the user types new text over old, the new text replaces the old on a character-by-character basis. This facility is most useful for mistyped characters. The user simply types the correct character over the wrong one. If the user mistypes a word, he may back up to the start of the word with one cursor step, then continue typing from that word as if the error never occurred. Successive characters will eliminate the incorrect word.

For the user to eliminate an extra character typed by mistake, he need only step the cursor over it in the Erase mode and the character is deleted.

***Text Insertion.*** Sometimes the user wishes to insert new text into already existing text. There are two control keys which do that for him:

- Begin text insertion
- End text insertion

Any text typed between these two keys is inserted at the current cursor position for insertion into a previously entered character string. The exact display strategy for allowing insertion of text is highly terminal-dependent. This is discussed further in Section 5.

### *Cursor Motion Commands*

There are control keystrokes to move the cursor either backwards or forwards in the current input character string. It is possible to move the cursor by large or small steps. It is possible to have the cursor erase or not erase the text which it passes over.

In effect, there are three different modes which characterize the way the cursor can move: step unit, direction of motion, and erasing or not erasing.

***Step unit.*** The unit of motion may be any of the following:

- A character.
- A lexical unit - i.e., up to the next punctuation sequence.
- A line.
- End of original insert - i.e., back to the end of the text being inserted (when this is defined).

***Direction of motion.*** The cursor might be moved backwards or forwards in the text.

***Erase or Do Not Erase.*** The cursor passes over text. When in the Erase mode, that text is erased; when not in the Erase mode, it is kept. The Erase mode is fleeting; it is maintained for only one cursor movement. The user is protected from accidentally erasing correct text after entering the Erase mode to erase an error. The actual keys assigned to specific functions are determined by the physical terminal used. Thus the function keys are named but not assigned codes in the following table.

| *KEY* | *FUNCTION* |
|---|---|
| Erase | Enter the Erase mode. (Enter non-erase mode after the next action.) |
| Reverse | Enter Backward mode. This is the base mode. The Input Interface is in Backward mode at the start of each new input. |
| Forward | Enter Forward mode. |
| Move | Step the cursor in the mode direction by one character (and erase that character if in Erase mode). |
| Move word | Step the cursor by one word i.e., to the next punctuation character sequence in the mode direction (and erase all characters passed over if in Erase mode). |
| Move line | Step the cursor to the end of the current line in the mode direction (and erase all characters passed over if in Erase mode). If the cursor is at the end of the current line, step, in the mode direction, to the end of the next line. |

The Input Interface editing functions are a subset of those provided by the Editor. The Editor is upward-compatible from the Input Interface. This service-wide consistency provides uniformity and ease of use throughout

## *CURSOR EDITING (CORRECTING) AND STYLE CHANGES*

This forms the core of the text-editing facilities by extending the Corrector's horizontal (intraline) cursor editing into two dimensions. The user performs after-the-fact correcting (that is, editing) on existing multiline text simply by moving

through the text and performing correcting functions. The dynamic display of the editing pointer in the text (represented by the cursor) is absolutely necessary for this kind of editing.

Correcting provides fine vertical movement and horizontal movement as well as function-key Insert/Delete. It is ***always*** available in Edit mode. The cursor moves by character locations, ignoring empty screen locations (though not ignoring explicit blanks). This does not mean that blank lines are not shown, but backspacing the cursor from one line to the last one does not involve backing through right-edge margin space on the previous line.

The Corrector's cursor movements are further restricted by only allowing those that are meaningful in the context of what the user is typing. That is, if the user is typing a single-line command (all previous commands having been executed and finished), it is generally meaningless to "correct" the input by backing up to a previous line. Normally, then, the vertical cursor movement of the corrector can be disabled in cases where it makes no sense, causing a bell so that the user knows he is still getting a response.

The Edit mode can be invoked either implicitly, as when the user is typing a command, or explicitly, at the user's request (with the Edit command). It can be used either to create new text or to modify old text, normally after (or while) reading it. Whenever the user is in this mode, the body of text being edited acts as if it was just entered as an insertion. So the Corrector applies to it without distinguishing whether it is part of a new insertion or of an already existing body of text. The Corrector (in order to provide efficient correcting during insertion) requires minimal effort to change text as it is being typed. Applying this to already existing text makes it easy to modify text while moving around in it. This is unlikely to occur by accident, however, since the cursor movements needed for reading are independent of the correcting movements, so that there is no reason for the user to be "correcting" old text unless he actually wants to change it.

Text is further protected by several factors:

- There is an explicit "Edit switch" that the user must turn on to allow modifying existing text (this can be in the form of a command or an actual locking switch on the terminal if that can be provided).

- Ideally, the cursor itself should indicate whether editing (modification) is enabled (see Appendix B). If this is not possible, the Edit mode is at least indicated in a noticeable way on the screen.

- The Editor will always be working on a copy of the text. The service provides automatic checkpointing (see Ref. 7), allows backing up (via UNDO) to earlier intermediate versions, should the user do something disastrous and unintentional.

- The service provides document protection which prevents unauthorized modification of text.

# 4. *EDITING COMMANDS*

This section provides more detail on the higher level commands provided by the Editor. The beginning user should have no need for these commands, and the basic editing described above (Section 3) should account for most of the editing done with the service. However, these commands are necessary to provide the full power required of a text editor. The organization of commands follows the discussion of Editor-related tasks in Section 2. Note that the actual forms of all commands shown below are merely illustrative. The command language interface is not specified in detail until a careful study of the user community is completed.

## *REPLACE*

Certain kinds of style changes are most easily made using the REPLACE command. This has already been mentioned above in its simple form (see Section 2). It is a powerful and easy-to-learn command which allows the user to "REPLACE <string1> BY <string2>". Like the FIND command mentioned above, the REPLACE can be UNDONE or REDONE, so that the user can easily iterate through a number of similar replacements, examining each one in turn, by means of REDO.

Whenever it is applied, the REPLACE looks for <string1> in the text (with the same searching options and defaults as the FIND command described above in Section 3). If it fails to find any occurrences of <string1>, it reports failure, and leaves the cursor where it was before the user performed the REPLACE. If it finds <string1>, it shows the user what that occurrence would look like with the replacement made. He can either ***accept*** that replacement or ***reject*** it. In ***either*** case, the cursor is repositioned after that occurrence, so that if he repeats the command (e.g., with REDO) it will not find the same occurrence again, but will look for the next. If the user wants to reject the replacement, and stop the REPLACE without moving the cursor, he can simply ABORT

The REPLACE command also allows the user to specify that he wants the command to iterate for all occurrences in the text (with or without requiring his confirmation after each one). Note, however, that the user can achieve the same effect with REDO, as long as he is willing to confirm each replacement. Whenever REPLACE is performed iteratively, it counts all occurrences of the string to be replaced and reports how many there are to the user ***before*** it actually does the replacement. This helps the user catch situations where he might otherwise inadvertently change fifty occurrences of "the" to "this".

The REPLACE command can be UNDONE in two ways (the user is prompted for which one he wants when he asks for UNDO): either ***all*** effects of the last REPLACE can be UNDONE at once, or the user can step back through the iteration and UNDO selectively. (Of course, for a noniterative REPLACE, these cases are the same.)

In the following examples, the command forms shown are merely illustrative. They are not intended as examples of the actual command language used, since this is different for different target communities.

To allow ...ore flexibility with less typing, the simple REPLACE <string1> BY <string2> is supplemented by the form

REPLACE <left> [<TEXT>] <right> BY <replace-TEXT>.

Here, <left> and <right> are optional fields which do not get replaced, but merely provide context to identify <TEXT>. Thus the user can say

REPLACE the quick [brown] fox BY red.

and get the result: "the quick red fox". This eliminates the extra typing of the simpler form of REPLACE, which would require

REPLACE the quick brown fox BY the quick red fox.

The command also allows flexible matching of text with various special features in the <string1> (or <left> <TEXT> <right>) part. For example, there is an ANY-TEXT operator ("*") which can appear in the string to be matched. Thus the user can change

"This will probabiy be true" into "This is true"

by applying the command

REPLACE This [*] true BY is.

Similarly, there are Restricted-text operators which allow matching punctuation marks, separators (space, carriage return, etc.), numbers, and "special" characters ($,*,&, etc.). These operators are augmented as needed for the target community if necessary. There is a case-shift matching control like that described above for FIND, and a similar feature at the character-by-character level within the match field (<string1>). Finally, the service-wide Quote character can be used to look for special control symbols and characters (including itself).

Capitalization at the word level is broken into four case-classes of words:

- All capitalized (example: "WORD")
- Starts with capital (example: "Word")
- All lower case (example: "word")
- Mixed (example: "DoD")

In general, the automatic case-matching preserves the case-class of the item matched, when the replacement is performed. (For the mixed class, this is not generally possible, and the command asks for user interaction to specify what should be done.) Unless the *exact* form of the REPLACE is used, the command

REPLACE word BY item

will thus replace all of the following:

word BY item
Word BY Item
WORD BY ITEM.

Most of these features are not required most of the time by most users, but every user will occasionally come upon a particularly messy editing task which can only be performed by exploiting the full power of the REPLACE command (or else by painfully going through the text line by line to look for all occurrences).

### *SPELLING CORRECTIONS*

In addition to the REPLACE command, a special SPELLING command is provided which allows the user either to make single corrections or to cause all occurrences of a word in the text to have their spellings changed.

Aside from performing the replacement, this command also marks these changes as Spelling Corrections (for later reference by coordinators or authors, see Section 5 below), and it alerts the User Monitor [5] which considers entering this spelling error into the service-wide lexicon of misspellings and typographical errors (see also Ref. 3).

Whenever the user finds a mispelled word, he uses the SPELLING command to correct it. The command prompts him by asking if he wants all occurrences of the same error changed. If he answers "yes", the Editor proceeds to find and count all such occurrences. If there are relatively few (five or fewer), the Editor reports how many there are, and presents each one to the user for confirmation. If there are more than

five, the Editor reports the number and asks the user if he wants to confirm each one, or to have the Editor go ahead and do them. (This runs the risk of changing occurrences of "to" into "too" by mistake, and the like, but this is at the user's discretion. Here, as elsewhere in the service, the user is acknowledged as the source of intelligence in the interaction. The service is merely a helpful, tireless slave.)

## *FORMATTING*

The Editor provides all document-preparation facilities for the service. The formatting philosophy is

1. Where possible, text is normally displayed in a formatted form, whether it is a finished document or in the process of being created or edited.

   Line boundaries, filling and justification, tabs, etc. are processed and output properly for the particular output device being used. (Note that line printers should be able to simulate terminal characteristics or must be as compatible as possible; however, when this i not the case, the only problem is that hard copy does not correspond to soft copy, though both are properly formatted.)

   The ideal is to keep a default display of one page of text on the screen during editing, as context, and to update this page as required (filling and justifying, etc.) when the user edits, so that he can always see exactly what he would see if he were to produce an immediate hard copy from the text. (This is terminal-dependent. See Appendix B.) This requires (in general) reprocessing text back to the previous paragraph every time anything is changed, though there are some heuristics that can reduce this amount of work (e.g., if th user changes "MONDAY" to "FRIDAY", keeping the lengths of words constant, the formatting stays the same).

2. The Editor produces acceptable document format without the user's having to do ***anything at all*** about explicit formatting. Reasonable assumptions (described below) are built in, to allow breaking lines only at word boundaries, treating mutliple blank lines as paragraphs, etc. The intent is that ***any*** input text will come out automatically as a reasonably formatted document.

3. The user is allowed to perform "critical" formatting,, overriding the automatic formatting to control the appearance of the final document by "formatting by effect" (using the Literal Format mode described below), essentially "drawing" the format on the screen.

In order to produce formatted documents with no effort on the user's part, the Editor embodies certain natural assumptions about the way most users type. These are verified for a given target community, and alternate assumptions may be incorporated instead with minimal effort, but the strategy given below is considered a reasonable first approximation.

As the user types text, he is never required to consider the width of the line (of the terminal he is using or of the output device he intends the ouput for, if this is different from his terminal). Thus he never *has* to worry about typing carriage returns or about breaking words in the middle. The Editor automatically inserts carriage returns at word boundaries, keeping a sufficient right margin to insure that this can be done in all but the most extreme cases. (The service is intended for users who are trying to get work done and so are assumed not to be interested in typing text like "jkjkjkjkjkjkjkjkjkjkjkjkjkjkjkjkjkjkjkjk" just to see how the Editor handles it. When such cases *do* occur, the line is simply broken (with a hyphen) at the extreme right edge of the screen.)

When the user does type carriage returns, they are respected in two ways. First, they generate new lines during input. That is, the Editor responds by showing the effect of a carriage return when the user types one. Second, they are saved as part of the text. Though the normal Automatic Format mode performs filling and justification, and treats carriage returns, spaces, and tabs as equivalent, the user can invoke the Literal Format mode (described in detail below), in which the carriage returns he originally entered are again treated as carriage returns.

The use of two (or three) carriage returns is considered a request for a new paragraph, and is treated this way in the Automatic Format mode, while a string of (more than three) carriage returns is considered a request for a new page in Automatic Format mode, when this makes sense for the output device being used.

Similarly, a string of leading spaces or a tab at the beginning of a line establishes a standard indentation (even if the user doesn't count them carefully), and a long string of spaces or tabs causes a line to be centered. In Automatic Format mode, when output is produced, tabular data can usually be recognized (by the sequence of new lines and indentations with which they were entered), and a justified table is output, with decimal points alligned for numeric data, where feasible.

The intent of the Automatic Format mode is to alleviate the need for the user to have to think about format *most* of the time. The processing performed is not complicated for the most part, and does not produce startling results. Most of the time, the user is unconcerned with the exact format of the output, and reasonable results are

accepted in return for the reduced effort of not having to pay any attention to formatting. The conventions used for Automatic Format mode (such as those suggested above) are established for a particular user community after studying the equipment and procedures currently in use, and are chosen to be as natural as possible, so that users never have to think about them.

However, occasions still arise which require precise control of format and the overriding of such automatic conventions as those assumed in the Automatic Format mode. For these cases, the Editor provides the Literal Format mode to allow the user to specify formats explicitly.

In later versions of the service, the Editor will interface to a document-preparation processor (such as are ubiquitous in data processing circles) in order to provide sophisticated formatting control. The present version, however, intended as it is for the military message processing environment, does not warrant complex format processing in the usual sense. It therefore provides the user with the simple option of typing text on his terminal screen exactly as he wants it to appear in output, and then inhibiting the Automatic Format processing whenever that particular document is output. In order to provide somewhat more flexibility, this *literal* mode can be turned on and off within a document, allowing parts of the document to be formatted automatically and other parts to be output exactly as they were input.*

### *RESTRUCTURING*

Occasionally the user needs to rearrange text, for instance by moving a paragraph to another place in the document, exchanging two sections, copying a sentence from one place to another, etc. These tasks are difficult to perform without a special command, since they would otherwise require large amounts of retyping.

The Editor provides the MOVE command to perform this task. The body of text to be moved can be selected in one of two ways, both of which use the cursor controls to point out a place in the text: either the user names the amount of text to be moved

---

* Military messages involve many strictly defined formats [1, 8]. These are composed by means of the user's interacting with the message processing service Functional Module itself [4], which simply leads the user through the various fields and components of the required format, and composes these parts into the finished, formatted message. The Editor is merely used to enter text for each of these fields, and so is not responsible for producing the actual format of the final message.

(Word, Sentence, Paragraph, etc.) and points the cursor anywhere in the text to be moved, or he uses the cursor to select the start and end of the text to be moved. The cursor is then positioned once more to mark the place where the text is to be inserted. The MOVE command can be REDONE to avoid having to repeat the process of picking up the same text. Thus, the user can pick up a sentence, MOVE it back to where it came from (by leaving the cursor where it is) and then REDO and select another location with the cursor, to copy the sentence.

The MOVE command only holds onto the text picked up for the duration of the command (though it can be REDONE). Often, however, the user wants to pick up part of a sentence, rewrite the rest, and then put back the part he picked up earlier. To allow this, the MOVE command has an alternate form, consisting of two separate commands: PICKUP and PUT. These perform the two halves of the MOVE operation, and allow other commands to be executed in between, without losing the text picked up. Performing the sequence PICKUP, PUT is identical to the single command MOVE.

Like the PLACEMARK command described above (see Section 3), the PICKUP and PUT commands have an optional argument which is normally null. By supplying a name (or number) to the PICKUP and PUT, the user can name the text picked up, and can juggle several pieces of text, putting them back together (or in different places) with PUT commands.

## *SPECIAL-PURPOSE SUB-EDITORS*

In using the message service, whether filling in specified fields of messages or entering free text, the user may require special editing capabilities for special kinds of data. In order of increasing specificity, the user may be dealing with

- Text (about which the service assumes nothing)
- Names (or titles) of people
- Dates
- Restricted vocabulary (such as security classifications)

The Editor is generally used to deal with the first class, namely free text. This can be anything the user wants to type, and the service can make no assumptions about it, and so cannot provide additional functions beyond spelling, style correction, etc. However, each of the other classes presents additional constraints on what the user may type, and therefore the Editor can provide additional functions in each case. In

some cases, the service knows in advance that the user is entering data of one of these classes, e.g., when the user is creating a message and the message processing service Functional Module [4] asks the user to fill in a field that requires a date. In other cases, the user is in the middle of entering free text, and he wants to enter a date as text: here the Editor allows the user to ask for the same date-handling capabilities he is used to getting when the service knows he is entering a date.

The functions provided for each of the last three classes above can be thought of as subeditors and are described below (free text which is handled by the Editor itself). In each case, the HELP and "?" functions (described in Refs. 3 and 6) allow the user to ask for all allowable inputs or for an explanation of what he is allowed to enter.

***Names and Titles***

In many cases the user wants to enter the name of a person or a title. This occurs when specifying addressees, coordinators, etc., and it may also occur when the user is typing text and wants to refer to someone in the body of his text. The service cannot be expected to know all possible names that the user might enter, but it can provide help with names it does know about, such as other users of the service or names the user has told it about in the past.

When the service knows that a name is being entered (as when the user is entering a list of addressees), it automatically provides the user with the facilities of the Name Sub-Editor. For other cases (as when the user types a name in the middle of free text), the user can enter a NAME command to invoke the Name Sub-Editor.

The Name Sub-Editor provides the following capabilities:

- User-defined abbreviations (such as first names) can be supplied.
- Initials are accepted where they can be recognized by the service or have been supplied by the user in the past.
- Names can be expanded to give full names (including initials), titles and ranks.
- Titles can be expanded to give names and ranks.
- Ranks of organization-oriented terms (like "the Colonel", "my boss" or "my secretary") can be expanded to give names and ranks.

Note that in all cases, invoking the Name Sub-Editor allows examining the resulting name before entering it as text. This means that the user can invoke this Sub-Editor simply to check on a name (for instance, to look up who J6 is, by expanding that title into a name) and can then return to the text without actually inserting that name.

Note also that whenever the Name Sub-Editor is invoked, the user is assuming that the service knows the name he is entering. (If the service does not know the name, then the Name Sub-Editor cannot perform any useful function, since a name can potentially be any arbitrary string of text.) The service interacts with the user to try to match the name he is entering with one that the service already knows, and if this is impossible, the service asks if the user wants it to remember this name for future reference.

***Dates***

This is an even more restricted set of possible entries, since every possible date is known in advance. The Date Sub-Editor allows:

a. Any of a number of forms for specifying an actual date:

   September 19, 1974
   Sept. 19, 1974
   19 September 1974
   9/19/74
   9-19-74

b. Giving the day of the week for any date supplied

c. Expanding relative dates:

   today
   yesterday
   the third Sunday in October
   last Wednesday
   etc.

d. Accessing a (perpetual) on-line calendar:

In all cases, defaults are supplied where necessary.

The year defaults to the current year if it is not supplied.

The Date Sub-Editor accepts any reasonable spelling or abbreviation to the names of months or days of the week.

Ranges on dates are checked to prevent entering September 31 or Feb. 29, 1974.

***Restricted Vocabulary***

This class includes all cases where the user is entering one of a finite number of choices known to the service, such as Security Classification, Special Handling, Message Type, etc. Here, the service knows exactly which inputs are legal, and so it attempts to match the user's input with one of the allowed ones. It can supply a menu of allowed inputs when asked to by the user. The user can ask for descriptions of any of the allowed inputs by requesting Help from the Tutor [6].

# 5. *ISSUES*

This section addresses a number of issues related to the message processing environment for which the Editor is designed. In particular, multi-author and coordinatic issues are discussed, and motivations are given for some of the strategies presented above, with respect to formatting and cursor editing.

## *COORDINATION*

Though the Editor is designed to look the same to the user no matter what text-handling job is being performed, there are certain special requirements placed on it by the military message processing environment.

The message processing Functional Module [4] handles the restriction of different classes of coordinators to different kinds of commenting or editing, and also leads the coordinator through the message on a per-field basis, so that he can comment in detail on fields other than the body of the message. The Editor simply provides the functional capabilities for commenting on and editing fields of messages.

In particular, in order for a coordinator to edit a message in a way which will be intelligible to other coordinators and to the author (see Ref. 4), the Editor provides coordinators with several special commands to categorize the changes made.

On the other side of the coordination process, the Editor allows the author to examine various categories of changes made by coordinators, so that he may select which, if any, to incorporate into the final message.

### *Coordinator Options*

Whenever a coordinator edits a message, he is really making suggestions to the author (and to subsequent coordinators) as to how the message should look. He does this by working on a copy of the orignal message, entering whatever changes he feels should be made. He has at his disposal all the editing facilities of the Editor and, in addition, he has the capability of identifying the changes he makes under several categories. The author (or other coordinators) can in turn look for changes under specific categories (see Examination Options).

Whenever the user edits a message being coordinated, the Editor keeps track of what *types* of changes he makes. The user can also select one of several *categories* to alert the author (and other coordinators) to the significance of the changes he is making. Each change which the user does not explicitly put into a category is placed by default into the *minor* category.

The CHANGE-TYPES are:

*Spelling Error.* Every time a coordinator uses the SPELLING command, the Editor records the command itself and each of the changes made by it, under this category. Though the service itself does not enforce the convention, coordinators are generally assumed to have verified their own changes so that the author need not repeat their efforts by checking everything they have done.

*Style (Rewrite).* This category is used for general rewriting changes. These can involve arbitrary rearrangements, insertions, deletions and changes to text. When a coordinator makes a *style* change, the Editor automatically marks as changed the unit of text that was involved. This allows the Editor to display the changed units to the author on request (see below for Viewing Options). However, a small unit (e.g., word) is always shown in context (e.g., the line containing the word) when displaying changes.

*Structure (Format).* These are changes consisting primarily of physical rearrangements of paragraphs, sections, etc. (e.g., interchanging sections "1.3" and "4.1"), or which affect the format of the message. They are distinguished from style changes because they represent changes in the organization of a message rather than in its writing style. They also generally require more context to be shown (or alternate display strategies) when being viewed by the author (see below under Viewing Coordination/Edit Changes).

The CHANGE-CATEGORIES allow the coordinator to tag his changes as being more or less important. The author can then examine changes by category. The categories are "nested" in the sense that all CRUCIAL changes are also CONTENT (and MINOR) changes. MINOR changes really include *all* changes.

The CHANGE-CATEGORIES are:

*Minor.* This is the default category. A coordinator may put changes into this class simply by refusing to specify categories for them. Included in this class are all changes in the CONTENT and CRUCIAL categories as well. This insures that the author does not miss more important changes when examining minor ones.

***Content.*** Here the coordinator calls the author's attention to serious changes, affecting the sense of the message. All CONTENT changes are also automatically categorized as MINOR changes. (Note that this category also includes all changes in the CRUCIAL category described below.)

***Crucial.*** This category is used for changes which the coordinator feels are major changes to the sense of the message, inaccurate statements, objections, etc. It can be used to point the author to those parts of the message which caused the coordinator to sign off OK?, OK-, or NoGood (see Ref. 4). All CRUCIAL changes are also automatically categorized as CONTENT and MINOR changes.

***Examination Options (Viewing Changes)***

When the author examines a message that has passed through a coordination phase, he has a number of options for viewing suggested changes made by coordinators. Coordinators also have a subset of these options available. These options are included under viewing.

Viewing is the ability to inspect all message fields. While viewing, the user has the ability to see all previous comments and edits (hereafter called annotations) incorporated into the message fields (either marked to delimit changes or unmarked), with or without identification of the users who made them. In viewing, the user may choose in which manner he wishes to view the various annotations, and the service will provide reasonable defaults for those left unspecified.

The CHANGE-TYPES and CHANGE-CATEGORIES described above give the author (and other coordinators) great flexibility in selecting what changes to look at. The message service also distinguishes the following classes of annotations:

> General Comments - These are contained in the comment text item subfields of each reviewer entry (see Ref. 4). They can be assigned to any of the CHANGE-CATEGORIES discussed above.
>
> In-Field Comments - These comments are within any message field. They can only be created by advisors and authors. They include the CHANGE-TYPES discussed above, and can be assigned to any of the CHANGE-CATEGORIES.

Readers and releasers can only make general comments. The reviewers' viewing default is to see the current state of the message. The authors' default is to see all changes. This is to enable and encourage the author to pass judgment on the suggested changes.

The above is discussed in detail in Ref. 4, and will not be repeated here. The CHANGE-CATEGORIES supported by the Editor under the Coordination mode permit the author to select any of the categories discussed in (1) above for any particular coordinator or over all coordinators. (In addition, he can select the MINOR category, which includes all changes.)

The following discussion addresses the author's (or coordinator's) options for viewing selected *types* of changes.

***Spelling Error.*** There are two options for viewing spelling changes. The actual changes made can be displayed one at a time, or the SPELLING command that was issued to make the changes can be shown instead. This latter option is used to see what the correction was, without actually verifying every instance that was changed. As when the SPELLING command is issued originally, the number of occurrences changed is also shown. When actual instances are examined, the original and changed forms are both shown. The exact display technique is highly terminal-dependent, and is discussed herein.

***Style (Rewrite).*** The Editor keeps track of the text affected by style changes, and displays them to the author (on request), while showing sufficient context for him to evaluate the changes. Again, the display techniques used are terminal-dependent.

***Structure (Format).*** These changes are not easily displayed on any device that can only show a limited amount of context (less than one typewritten page for most soft copy terminals). The Editor allows the author to examine the moved text in its old and new contexts. (Recognizing the difficulty of displaying certain types of restructuring, the Editor also provides hard copy output with moved sections of text marked as such.)

These viewing options give the author a powerful tool for considering which changes to incorporate. Some examples of the ways this can be used are

- Look at all CRUCIAL changes to the message,
- Look at all STYLE changes made by Col. Smith,
- Accept all SPELLING changes made by Captain Jones.

The author can thus delegate responsibility for certain tasks (such as proofreading) without even having to check over the results unless he wants to.

### *FORMATTING*

The strategy described above (Section 4) is intended to help the user produce formatted text with minimal effort. An initial set of assumptions or conventions was specified above. These are intended to produce reasonable formatting in most cases for most target communities. However, realizing that the proper conventions depend on many factors, including the work style and existing conventions of the target community, the formatting requirements of the end users, and the terminals used, the Editor is designed to allow redefining or overriding the default conventions discussed above. The Automatic Format mode can be disabled entirely, and the Literal Format mode used instead, if desired, though this is a retreat to manual formatting. Alternatively, the formatting conventions (e.g., use of multiple carriage returns to signal new paragraphs automatically) can be replaced by other conventions, or by more formal formatting tools. (As mentioned above, later versions of the Editor will interface to document-preparation processors capable of handling sophisticated formatting tasks.) The Automatic Formatting approach is considered well worth the attempt since it frees the user from a time-consuming and uninteresting aspect of creating documents.

### *CORRECTING (Cursor Editing)*

In this and the following section, it must be kept in mind that the service is intended to use soft copy terminals such as television-like (CRT) terminals. The Editor is not alone among the service elements in requiring the flexibility and relative speed of these devices, as opposed to hard copy terminals. It is assumed further that the user has access to output devices which produce hard copy when he needs it.

The use of cursor positioning and "direct" insert, delete, and overstrike editing is contrary to the techniques used in many existing on-line editors. It was chosen for the message service Editor because it relies more on self-explanatory keys on the terminal than on the user's memorizing a set of editing commands. The success of this approach is highly dependent on the terminal used (see Appendix B), but it has the advantage of requiring minimal training in the basic use of the Editor. As discussed above (Section 3), the Corrector functions are sufficient for most editing tasks. The user need only use the editing commands (Section 4) for relatively sophisticated editing, and these commands are kept to a minimum.

The actual display strategy used for inserting, deleting, and overstriking (replacing) text with the Corrector depends on the terminal used. The issues here are ones of bandwidth and buffering capacities, and ideal solutions are only possible with ideal terminals (see Appendix B).

There is no real question that what the user would like to see is simply the updated, corrected text, displayed in its current state at all times. That is, when he deletes a character from the middle of a word on the top line of a twenty-line screenful of text he would like the screen to flicker imperceptibly and to have the deleted character disappear, with the others closing in to fill the gap. (If the text is being formatted automatically, he would also like to see proper filling and justification performed instantaneously.) This is not practical with most terminals, though it can be approximated (without the reformatting) by some terminals which include "local" processing capabilities. The limiting factor is generally the time needed (with standard data transmission rates) to completely rewrite the screen with new text, starting at the point where the user changed it.

### *VIEWING COORDINATION/EDIT CHANGES*

There are further limitations which most existing terminals present in displaying changed text, as when an author is viewing suggestions and changes made by coordinators.

Here, the ideal is to display, side by side, the original and changed versions. Few terminals can provide this in a readable way. The only reasonable alternative, for certain classes of changes (e.g., long or complicated style changes, formatting, or restructuring changes) is to display both version one above the other (or one after the other in sequence), with suitable labelling. For other changes, a single copy of the text is displayed, with the changed items highlighted in one of several ways (depending on the terminals used). Candidate techniques for displaying such changes are reversed video (printing black-on-white instead of the usual white-on-black used by most soft copy terminals), bracketing the changed text with special brackets, underlining on the terminal, double intensity, etc. (Note that blinking is *not* considered, since it is highly distracting and hard to read.) One or more of these techniques are used by the Editor for displaying simple insertions, deletions, and replacements.

### *SUMMARY*

The Editor described above allows inexperienced or occasional users to perform most editing tasks using a minimal set of simple commands, relying on terminal controls and function keys for the most frequent operations. More sophisticated users can perform powerful editing functions with a handful of additional commands. In keeping with its role in the message processing service, the Editor is closely geared to the tasks of message authorship and coordination.

## *Appendix A*

## *EDITOR COMMANDS*

The following summarizes the Editor functions. The actual command names and forms are determined by command tables for the CLP, so that while the description below is semantically accurate, it is merely suggestive of the ultimate syntax seen by the user. The Editor is seen by all other processes as the function

EDIT (Modify, Call-type, Data-type, Destination, [Command-stream, Result])

| | | |
|---|---|---|
| Modify | ::= | TRUE / FALSE |
| Call-type | ::= | USER-EDIT / CALLED |
| Data-type | ::= | TEXT / USER-NAME / MSG-ID / DATE-TIME |
| Destination | ::= | file-address |
| Command-stream | ::= | { commands passed by calling module in CALLED mode } |
| Result | ::= | { passes result in CALLED mode } |

The Modify switch allows calling the Editor in a restricted mode, so that no modification to the accessed text is allowed.

CALLED is used for calling the Editor from some other module, performing the specified "command-stream," and returning a "result." The User-Edit mode is the normal mode with commands coming from the CLP and results going back through the CLP to the user.

All functions which move the cursor refresh the display so the user always sees his context. All function-key commands are noted by "!" The design of the Editor (and in fact of the entire service) calls for a "logical" function box with cursor-moving and other function keys. These functions may eventually be merged into the terminal, but at present they are provided by an actual function-key box, the Auxiliary Keyboard and Scope Multiplexor (AKSM).

All commands (unless noted otherwise) can be undone by !UNDO and redone by !REDO. The !REDO key will also serve as a confirmation if the command was expecting one. (A Command-Abort followed by !REDO will repeat the command without confirming the previous instance.)

The Corrector's cursor-moving capability is used in three ways:

1. To correct Editor commands in the command window before executing them.

2. To edit text displayed on the screen.

3. In conjunction with other commands, to allow selecting a position or string of text by means of entering a cursor position explicitly. In order to do this, the user moves the cursor to the desired location on the screen and then hits the !HERE function key.

The commands below are given in Internal Form first. Following the formal description, the User Form is given. If the latter is different from the former, square brackets ([]) designate an optional field.

### *KEYSTROKE (CORRECTOR) FUNCTIONS*

!EDIT (On/off switch)

When this switch is off, the Corrector controls (below) apply to the text in a read-only mode. That is, the user can look through the text, but cannot modify it.

MOVE-CURSOR (Direction, Amount, Erasing)

Direction ::= LEFT / RIGHT

Amount ::= CHARACTER / WORD / LINE / PARAGRAPH /

Erasing ::= TRUE / FALSE

### *User Form*

These are all function-key commands. Keys are as follows:

| | |
|---|---|
| !LEFT-CHARACTER | !RIGHT-CHARACTER |
| !LEFT-WORD | !RIGHT-WORD |
| !UP-LINE | !DOWN-LINE |

All of the above are repeating keys. They can be "shifted" by an ERASE key which moves the cursor and erases at the same time. When the ERASE key is depressed, the MOVE keys do not repeat. In addition, there are the following keys, which are not repeating and cannot be "shifted" to ERASE.

| | |
|---|---|
| !UP-PARAGRAPH | !DOWN-PARAGRAPH |
| !UP-SCREENFUL | !DOWN-SCREENFUL |

!ENTER/LEAVE INSERT-MODE (On/off switch)

When this switch is off, if the cursor is positioned in the text on the screen (outside the command window), typing any alphameric character causes the cursor to return to the command window to accept further commands.

!REFRESH-SCREEN

This reformats and redisplays on request (not usually needed, since screen is automatically reformatted frequently).

### *COMMANDS*

A number of the commands below, including FIND and REPLACE, position the cursor in the text being displayed (as opposed to a command being typed in the command window). These commands leave the cursor positioned in the text to allow Corrector changes to be made, or to allow selecting the location with the !HERE key.

When the cursor is positioned in the text shown on the screen, only the Corrector commands (including !INSERT) and the marking command !HERE are available without moving the cursor to the command window. When the user types anything else, the cursor automatically homes to the command window, leaving the location in the text marked. To select this marked location easily after typing further commands, the !HERE key can be hit when the cursor is in the command window and it will return the cursor to the last marked location. The Editor then asks for confirmation (either with a second !HERE or with Command-accept).

### *I. Moving the Cursor*

FIND (String,Type,Exact,Count)

Type ::= STRING / WORD

Exact ::= TRUE / FALSE

Count ::= TRUE / FALSE

A successful match always repositions the cursor and marks the end of the string found. Successive FINDs will find successive occurrences.

Defaults are: Type = STRING, Exact = FALSE, Count = FALSE.

If String is defaulted, the last Find-String is used (that is, the last string used in a FIND command that was successfully completed).

Case-independent matching is used ***unless*** the "Exact" mode is on. The Exact mode can be set explicitly (by "EXACT", "E", or "↑E"), but it is also set implicitly whenever "String" contains any upper-case characters. In order to override this when upper-case characters are entered by mistake, a "↑N" resets the mode to be Not-Exact.

If "Count" is true, FIND returns a count of how many occurrences there are in the buffer, and asks the user whether or not he really wants to see all occurrences (each with context).

***User Form***

[ Q ] [<SP>] FIND [ Q ] <SP> Str

<SP> = (space)

Str ::= string / <Lptr , Rptr>

<Lptr , Rptr> designates a pair of cursor-positions selected by the !HERE key.

Q ::= Qc , Qe , Qw (in any combination, any order)

Qc ::= COUNT / C / ALL / A

Qe ::= EXACT / E / ↑E (/ ↑N for "NOT-EXACT", the default)*

Qw ::= WORD / W / ↑W

Examples: FIND giraffes' necks
FINDEXACT Giraffes' necks
WORD FIND giraffe

PLACEMARK ([ Name ] , [ Ptr ])

Name ::= string / NULL

Ptr gives a single cursor location. Normally the !HERE key is used to select this location, but the user can also invoke a FIND command to select it.

This command places a temporary mark in the text the user is editing. The mark can be named (to allow several places to be marked at once). Marks can reside in several messages (or files or folders) at once to allow jumping back and forth between them.

If the user does not supply a name for a placemark, the default (null) name is used. Reusing the name of an existing placemark (e.g., the null name) erases the previous placemark with that name. Marks last only for the duration of the editing session.

If Ptr is not specified, the default is the current location of the text pointer.

***User Form***

[ PLACE / MARK ] [<SP>] [ MARK / PLACE ] ["name"]

Ptr ::= !HERE / ↑F

↑F invokes a Find command and uses the resulting cursor position as the selected location.

Examples: PLACEMARK
MARK PLACE "1"
PLACE "q"

---

* ↑W, ↑E, and ↑N can appear anywhere in the command, including within Str.

FINDPLACE ([ Name ])

Name ::= string / NULL

This command returns to the last place marked with this name (default is the null name) by the last PLACEMARK command.

***User Form***

FIND [<SP>] < PLACE / MARK > ["name"]

Examples: FINDPLACE
FIND MARK "q"

### *II. Changing Text*

REPLACE ([Left], Text, [Right], Replace-text, Type, Exact, Count)

Left , Right , Text , Replace-text ::= Str

Type ::= STRING / WORD

Exact ::= TRUE / FALSE

Count ::= TRUE / FALSE

Str ::= String / String Ops / Ops String / Ops String Ops

String ::= String / <Lptr , Rptr>

<Lptr , Rptr> designates a pair of cursor positions selected by the !HERE key.

Ops ::= Op / Op Ops

Op ::= * / $ / # / & / -

where

| | |
|---|---|
| * | any text |
| $ | any separator (space,CR,period,etc. ?) |
| # | any digit or string of digits |
| & | any special character |
| - | not some character or class of characters |

Defaults are: Type = STRING, Exact = FALSE, Count = FALSE.

Any of the text fields {Left}, {Right}, Text, or Replace-text can be defaulted to the last value used for the corresponding string in the last REPLACE command.

Every time a match is found, the old and new versions of the text in question are displayed and the user is asked for confirmation.

The Count switch behaves as in the FIND command: it causes the number of occurrences to be counted and reported before any replacements are made, then asks the user whether he wants to verify each replacement, to have them all done without his seeing them, or to abort.

The Exact mode for the REPLACE command* is like that of the FIND command, with a few more cases:

1. <Qe> or any upper-case characters in text force the entire command to be Exact.

2. Upper-case characters in {Left} or {Right} apply only to the match within Left or Right and do not force each other or the entire command into Exact mode.

3. Upper-case characters or ↑E in Replace-text forces the replacement to be Exact, but not the matching of text. (In this case, the command expansion indicates that only the replacement is Exact, and a second ↑E will force the entire command to be Exact so the user need not start over. Alternatively, the CLP provides the !FIX key, which allows editing the command itself, and so changing it to be Exact.)

---

* Like the FIND command, this command always marks the end of the matched or replaced string and leaves the cursor there. The command can be !REDONE on successive occurrences.

For details on the handling of upper and lower case, see the "case-classes" defined in Section 4.

***User Form***

[ Q ] [<SP>] REPLACE [ Q ] <SP> [{Left}] Text [{Right}]
<BY> Replace-text <End>

<SP> = (space)

Text, Replace-text, Left, Right ::= string / <Lptr , Rptr>

<Lptr , Rptr> designates a pair of cursor-positions selected by the !HERE key.

{left} , {right} define context for text, but are *not* replaced.

<BY> , <end> ::= Command-accept (but *not* CR -- ESC is ok)
[ Replace works across line-boundaries.]

Q ::= Qc , Qe , Qw (in any combination, any order)

Qc ::= COUNT / C / ALL / A

Qe ::= EXACT / E / ↑E (/ ↑N for "NOT-EXACT", the default)

↑W, ↑E, and ↑N can appear anywhere in the command, including within Str.

Qw ::= WORD / W / ↑W

Examples: REPLACE this is <BY> that was <end>
REPLACEW attitude <BY> altitude <end>
EXACT REPLACEALL thru <BY> Through <end>

SPELL (Incorrect, Correct)

Incorrect , Correct ::= string

This command performs the equivalent of

Replace (Incorrect, Correct, WORD, Not-exact, COUNT)

This will replace all occurrences of the word, regardless of case. Before making any replacements, it counts the number of matches and reports the number to the user (to help catch inadverter' errors affecting many words). It then asks the user if he wants to confirm each occurrence or if he wants them to be done without confirmation.

In addition, the Service Glossary is updated to look for this error, and the change to the message is noted as a Spelling Change. Both the "rule" itself (Incorrect,Correct) and the actual changes made are saved for subsequent Viewing.

Defaults: either of the strings Incorrect or Correct can be defaulted to the value of the corresponding string in the last SPELL command.

***User Form***

SPELL Incorrect <AS> Correct <end>

Incorrect , Correct ::= string

<AS> , <end> ::= Command-accept / CR / ESC / "."

Examples: SPELL thaanks <AS> thanks <end>
SPELL Usr <AS> user <end> (case is ignored).

PICKUP ( Type, Pointer, [ Name ] )

Type ::= STRING / WORD

Pointer ::= Ptr / <Lptr , Rptr>

Ptr gives a single cursor location. Normally the !HERE key is used to select this location, but the user can also invoke a FIND command to select it.

<Lptr , Rptr> gives a pair of cursor locations. Normally the !HERE key is used to select these locations, but the user can also invoke a FIND command to select each location.

Name ::= string

This command picks up the delimited text and saves it under the (possibly null) name. It also erases any string previously picked up under this name. The text can be put somewhere else with the Put command. Note that this command has scope across ASIDE commands, so that text can be picked up from one file (or message) and put into another. The text delimited is marked on the screen. The optional name allows several distinct pieces of text to be manipulated at once.

If Type = STRING, a pair of pointers is used to delimit the text. Otherwise a single pointer is sufficient, since the type of object has been named. (That is, with Type = WORD, a pointer anywhere within the word will PICKUP the entire word.)

If two PICKUP commands with the same (e.g., null) name are entered without an intervening PUT command (or if a PICKUP command is !REDONE), a warning is given to make sure the user is aware he is destroying the previously picked up text.

Ptr or Lptr can be defaulted to the current location of the text pointer.

***User Form***

[ Q ] [<SP>] PICKUP [ Q ] ["name"] Pointer

| | | |
|---|---|---|
| Q | ::= | WORD / W |
| Pointer | ::= | Ptr / Lptr , Rptr |
| Ptr | ::= | !HERE / ↑F |
| Lptr , Rptr | ::= | !HERE / ↑F |

↑F invokes a FIND command and uses the resulting cursor position as the selected location.

Examples (prompting is shown in single parentheses):

PICKUP (between) !HERE (and) !HERE

PICKUP (between) ↑F ((FIND the man)) (and) !HERE

PICKUP "first" (between) !HERE
(and) ↑F ((FINDWORD multiple))

PICKUPWORD (at) !HERE

PICKUPW (at) ↑F ((FINDW bout))

PUT ( Ptr , [ Name ] )

Ptr gives a single cursor location. Normally the !HERE key is used to select this location, but the user can also invoke a FIND command to select it.

Name ::= string

This command inserts the text picked up by the last PICKUP command with the same name. It requires only a single cursor position for inserting the text.

The text can be PUT several times, into several locations. After a PUT, the cursor remains at the end of the inserted text. Two successive PUT commands will therefore repeat a piece of text.

Ptr can be defaulted to the current location of the text pointer.

***User Form***

PUT ["name"] Ptr

Ptr ::= !HERE / ↑F

↑F invokes a FIND command and uses the resulting cursor position as the selected location.

Examples: PUT !HERE
PUT "first" ↑F ((FINDE banana))

SHOW ([ name ])

This shows the text picked up by the last PICKUP command using this name. The default for Name is null.

***User Form***

SHOW ["name"]

Examples: SHOW
SHOW "first"

MOVE ( Type, Pointer , Ptr )

Pointer ::= Ptr / <Lptr , Rptr>

Ptr gives a single cursor location. Normally the !HERE key is used to select this location, but the user can also invoke a FIND command to select it.

<Lptr , Rptr> gives a pair of cursor locations. Normally the !HERE key is used to select these locations, but the user can also invoke a FIND command to select each location.

This is almost equivalent to performing the sequence PICKUP, PUT. The user is prompted to supply cursor locations to delimit the source text and to specify the destination.

If Type = STRING, a pair of pointers is used to delimit the source text. Otherwise a single pointer is sufficient, since the type of object has been named. (That is, with Type = WORD, a pointer anywhere within the word will select the entire word.) The destination is always selected by a single pointer.

Ptr or Lptr can be defaulted to the current location of the text pointer, or the user can default the entire source text location, forcing the Move to use the last Moved Text and to copy it into the new destination. In this case, nothing is deleted. There are no names for Moves, since they have essentially no scope, except for this special case (of defaulting the source text). This defaulting need not be done immediately after the previous Move. There is never any interaction between text saved by the PICKUP command and text handled by the Move command.

***User Form***

[ Q ] [<SP>] MOVE [ Q ] Pointer Ptr

Q ::= WORD / W

Pointer ::= Ptr / Lptr , Rptr

Ptr ::= !HERE / ↑F

Lptr , Rptr ::= !HERE / ↑F

↑F invokes a FIND command and uses the resulting cursor position as the selected location.

Examples (prompting is shown in single parentheses):

MOVE (from between, !HERE (and) !HERE (to) !HERE

MOVE (from between) ↑F ((FIND giraffe)) (and) !HERE
(to) ↑F ((FIND banana))

MOVEW (from) !HERE (to) !HERE

MOVEWORD (from) ↑F ((FIND giraffe)) (to) !HERE

***Miscellaneous Commands***

KIND-OF-CHANGE (Type)

Type ::= CRUCIAL / CONTENT / MINOR

This command lets the user tag his changes for the purposes of coordination. His changes can then be reviewed by Type via the View command. Note that the categories are nested so that Minor changes include all Content and Crucial changes as well. This insures that the more important changes are not missed when viewing the less important ones. The default is (therefore): Type = MINOR.

***User Form***

KIND [<SP> / -] [OF [<SP> / -] CHANGE] [IS / = / <SP>] Type

Type ::= MINOR / M / CONTENT / C / IMPORTANT / I

Examples: KIND OF CHANGE IS CONTENT
KIND=I

VIEW (Change-type,Kind-of-change,By)

Change-type ::= Spelling / FORM / OTHER / ALL

Spelling ::= RULE / INSTANCES

Kind-of-change ::= MINOR / CONTENT / IMPORTANT

By ::= <reviewer-name> / ALL ]

This command allows selectively seeing changes made by reviewers. Selection can be on the basis of ***who*** made the change, what ***kind*** of change it was, and what ***type*** of change it was.

Defaults are to see all changes.

***User Form***

VIEW Change-type Kind-of-change By

Change-type ::= SPELLING / S / RULE / R / INSTANCES / I
FORM / F / OTHER / O

Kind-of-change ::= MINOR / M / CONTENT / C / IMPORTANT / I

By ::= <reviewer-name> / ALL ]

BEGIN-LITERAL-FORMAT (Ptr)

This command allows turning off any automatic formatting, and insures that text will appear exactly as it is entered from this point on in the message.

Ptr can be defaulted to the current location of the text pointer.

***User Form***

BEGIN [ - / <SP>] LITERAL [ - / <SP>] FORMAT

Ptr ::= !HERE / ↑F

↑F invokes a FIND command and uses the resulting cursor position as the selected location.

END-LITERAL-FORMAT (Ptr)

This command turns automatic formatting back on.

Ptr can be defaulted to the current location of the text pointer.

***User Form***

END [ - / <SP>] LITERAL [ - / <SP>] FORMAT

!SPECIAL-EDIT (Data-type, String)

Data-type ::= USER-NAME / MSG-ID / DATE-TIME / <one-of>

When the user is filling in certain fields of messages, he is not dealing with simple free text. In these cases, the Function Module has alerted the Editor as to what kind of data to expect. In these cases, the Editor supplies the recognition, validation, and option-display facilities appropriate to the data-type being entered.

When the user is entering free text (as in the body of a message), he can also access these special-editing facilities for entering a special data-type, but he must first alert the Editor that what he is entering is data of one of these types (which the Editor is otherwise incapable of distinguishing from free text).

The !SPECIAL-EDIT key alerts the editor for one of these special data-types. In this mode, the user has several options:

? equivalent to CLP "?" command: shows options for this data-type.

$ recognition: expands input if valid

<brk> asks Function Module to validate the data. If not a valid entry of required data-type, user can edit and try again. Any break character for this data-type is accepted.

<end> exits from the special mode, leaving the result in the text *if* it was valid (otherwise the text is not affected).

In the cases of USER-NAME, MSG-ID, and DATE-TIME, the user simply enters a string which the Editor passes off to the Function Module to validate and (possibly) recognize.

USER-NAME deals with names and titles.

MSG-ID deals with message names/id's.

DATE deals with dates and times.

Defaults: the last data-type encountered and the last value of that data-type are used if data-type or string are defaulted.

***User Form***

!SPECIAL [ -EDIT ] data-type string

## *Appendix B*

## *TERMINAL CONSIDERATIONS*

Many aspects of on-line editing (and message processing in general) are strongly affected by the terminals used. The military message processing service assumes soft-copy (e.g., CRT) terminals, but within this broad category, there are a number of requirements and desiderata.

The following features are considered essential:

- High bandwidth (at least 1200 baud).
- Cursor control keys (single keys for "up, down, left, right, home", which send codes out to the computer). These should repeat individually, if possible, or by means of a repeat button. (Note that some alternative means of positioning the cursor (such as the SRI Mouse) are also acceptable here, and may in some cases be preferable to cursor movement buttons.)
- Corresponding cursor functions (that can be sent from the computer to the terminal).
- Upper/lower case (with shift lock).
- Programmable function keys (that is, keys which send codes that are neither alphameric characters nor assigned functions).
- At least one bell or similar "alert" mechanism (two distinct ones would be desirable).
- Reverse video (black-on-white printing), two intensity levels, or underscoring (as many of these alternatives as possible). This is necessary for highlighting text changes, etc. Blink may also be useful for alerting the user to high priority items on the screen, but is ***not*** a substitute for one of the above, since it is too annoying.

- In addition, the following are desirable:

  - Erase-to-end-of-line (sendable to terminal).

  - Erase-to-end-of-screen (sendable to terminal).

  - Cursor position read command.

  - Tab character.

  - Ability to disable any purely local keys.

  - Multiple cursors/bugs (a symbol that can be left on the screen to mark something - either under the line or by reversing video).

## *CURSOR*

***Read Mode.*** In Read mode, underscore to left of the top (or bottom) line of the page. Cursor always sits at the left of a line in this mode.

***Edit Mode.*** In Edit mode, underscore under the last character typed, with an arrow pointing up at the right to show where the insert goes. This insures that deletions always affect the character (or word) over the cursor, whereas insertions always be to the right of the cursor (preferably shown by the arrow). The cursor itself in this ideal case indicates that the Edit mode is on, by the presence of the insertion arrow.

## *FUNCTION KEYS*

The following service-wide functions are always available, preferably as single, labelled keys:

HELP
ABORT
UNDO
REDO (last command with same arguments)
RECALL (last command to allow editing and reissuing it)
CONTINUE (after an ABORT, etc.)
ALERT-CLP (gets the attention of the CLP)
"?" (for CLP, to ask for options available at any point)
SAVE (text and state)

When function keys are disabled (in certain contexts), the terminal should indicate this by either physically locking that key (selectively), beeping, or having a light on the key.

The Edit switch should be indicated either by a light or by the shape of the cursor itself (as suggested above). The cursor might also be made to blink whenever the user was in Edit mode as a warning.

## *REFERENCES*


1 Oestreicher, D. R., J. F. Heafner, and J. Rothenberg, ***CONNECT: A User-Oriented Communications Service***, presented at ACM Annual Conference, San Diego, Calif., November 1974.

2 Ellis, T. O., L. G. Gallenson, J. F. Heafner, and J. T. Melvin, ***A Plan for Consolidation and Automation of Military Telecommunications on Oahu***, ISI/RR-73-12, May 1973.

3 Abbott, R. J., ***A Command Language Processor for Flexible Interface Design***, ISI/RR-74-24, September 1974.

4 Tugender, R., and D. R. Oestreicher, ***Basic Functional Capabilities for a Military Message Processing Service***, ISI/RR-74-23, May 1975.

5 Heafner, J. F., ***A Methodology for Selecting and Refining Man-Computer Languages to Improve User's Performance***, ISI/RR-74-21, September 1974.

6 Rothenberg, J., ***An Intelligent Tutor: On-line Documentation and Help for a Military Message Service***, ISI/RR-74-26, May 1975.

7 Mandell, R. L., ***An Executive Design to Support Military Message Processing Under TENEX***, ISI/RR-74-25 (in preparation).

8 Fleet Operations Control Center, Pacific, ***Automatic Outgoing Message Processor System User's Guide***, FOCCPAC Document CM-02, 1973.

## *BIBLIOGRAPHY*

Engelbart, D. C., Watson, R. W., and Norton, J. C., "The Augmented Knowledge Workshop," ***AFIPS Proceedings***, National Computer Conference. Vol. 42, pp. 9-21, June 1973.

Contains an outline of the general philosophy of ARC, including research goals and strategies as well as historical details. Also has a bibliography of ARC reports as well as influences on our work.

Engelbart, D. C., "Design Considerations for Knowledge Workshop Terminals," ***AFIPS Proceedings***, National Computer Conference. Vol. 42, pp. 221-227, June 1973.

Contains description of the types of terminals and terminal interactions required and available in the ARC environment and the motivations for their development, as well as an extensive annotated bibliography.

Irby, Charles H., "Display Techniques for Interactive Text Manipulation." ***AFIPS Proceedings***, National Computer Conference. Vol. 43, pp. 247-255, May 1974.

Discusses the ARC model for two-dimensional text tools based on interactive display terminals and presents the primitives provided by the conceptual display terminal interface to an application program.

Andrews, Donald I., "Line Processor--A Device for Amplification of Display Terminal Capabilities for Text Manipulation." ***AFIPS Proceedings***, National Computer Conference. Vol. 43, pp. 257-265, May 1974.

Describes the Line Processor interface created at ARC to permit the use of inexpensive alphanumeric video display terminals with NLS in the model outlined in Irby's paper.

Engelbart, D. C., and English, W. K., "A Research Center for Augmenting Human Intellect." *AFIPS Proceedings*, Fall Joint Computer Conference. Vol. 33, pp. 395-410, May 1974.

General presentation of ARC and NLS functions as they existed in 1968. Much remains valid today, though of course our more recent project reports and internal documentation supersede large sections dealing with implementation details.